

# Fault Tolerant Network Constructors \*

Othon Michail<sup>1</sup>, Paul G. Spirakis<sup>1,2</sup>, and Michail Theofilatos<sup>1</sup>

<sup>1</sup>Department of Computer Science, University of Liverpool, UK

<sup>2</sup>Computer Engineering and Informatics Department, University of Patras, Greece

Email: {Othon.Michail, P.Spirakis, Michail.Theofilatos}@liverpool.ac.uk

## Abstract

In this work we examine what graphs (networks) can be stably and distributedly formed if adversarial crash failures may happen. Our dynamic graphs are constructed by fixed memory protocols, which are like population protocols but also allow nodes to form/delete links when pairwise interactions occur (Network Constructors). First, we consider standard Network Constructors (i.e. without fault notifications) and we partially characterize the class of such protocols that are fault-tolerant. We show that the class is non-empty but small. Then, we assume a minimal form of fault notifications (N-NET protocols) and we give fault-tolerant protocols for constructing graphs such as spanning star and spanning line. We show a fault tolerant construction of a Turing Machine  $M$  that allows a fault tolerant construction of any graph accepted by  $M$  in linear space with a population waste of  $\min\{n/2 + f(n), n\}$  (due to the construction of  $M$ ), where  $f(n)$  is an upper bound on the number of faults. We then extend the class of graphs to any graph accepted in  $O(n^2)$  space, by allowing  $\min\{2n/3 + f(n), n\}$  waste. Finally, we use non-constant memory to achieve a general fault-tolerant restart of any N-NET protocol with no waste.

**Keywords:** network construction; distributed protocol; self stabilization; fault tolerant protocol; dynamic graph formation; population; fairness; self-organization;

## 1 Introduction and Related Work

In this work, we address the issue of the dynamic formation of graphs under faults. We do this in a minimal setting, that is, a population of agents running *Population Protocols* that can additionally activate/deactivate links when nodes meet. This model was introduced in [1], called *Network Constructors*, and is strongly inspired by the *Population Protocol* (PP) model [2] and the *Mediated Population Protocol* (MPP) model [3]. Population Protocols run on networks that consist of computational entities called *agents*. One of the challenging characteristics is that the agents have no control over the schedule of interactions with each other. In a population of  $n$  agents, repeatedly

---

\*All authors were supported by the EEE/CS initiative NeST. The last author was also supported by the Leverhulme Research Centre for Functional Materials Design. This work was partially supported by the EPSRC Grant EP/P02002X/1 on Algorithmic Aspects of Temporal Graphs.

a pair of agents is chosen to interact, and they update their states based on their previous states. In general, the interactions are scheduled by a *fair scheduler*. When the execution time of a protocol needs to be examined, a very common example of a fair scheduler is the selection of pairs at random. The main difference between PPs and Network Constructors is that in the PP (and the MPP) models, the focus is on computation of functions of some input values, while Network Constructors are mostly concerned about the stable formation of networks satisfying some graph property. Fault tolerance has now to do additionally with the graph configuration, thus, previous results on self-stabilizing PPs and MPPs [4, 5] do not apply here.

In [1], Michail and Spirakis give protocols for several basic network construction problems, and they prove several universality results by presenting generic protocols that are capable of simulating a Turing Machine and exploiting it in order to stably construct a large class of networks, in the absence of crash failures.

In this work, we examine what networks can be stably formed if adversarial crash faults may exist. Here, *adversarial crash faults* mean that an adversary knows the rules of the protocol and can select at any time some node to remove from the population. We assume that the faults can only happen sequentially, that is, in every step at most one fault may occur.

A main difference between our work and existing self-stabilization approaches is that, due to constant local memory combined with possibly unbounded (e.g. linear) connections with other nodes, the nodes cannot distinguish whether they still have some activated connections with the remaining nodes or not, after a fault has occurred. This difficulty is the reason why it is not sufficient to just restart the state of a node in case of a fault, hence existing self-stabilization approaches cannot be directly applied here [6, 7]. In addition, in contrast to previous self-stabilizing approaches [8, 9] that are based on *shared memory* models, two adjacent nodes can only store 1 bit of memory in the edge joining them, which denotes the existence or not of a connection between them.

Angluin et al. [12] incorporated the notion of self-stabilization into the population protocol model, giving self-stabilizing protocols for some problems such as leader election. They focus on the goal of stably maintaining some property such as a legal coloring of the communication graph, or having a unique leader.

A previous work of Delporte-Gallet et al. [10] studies the issue of correctly computing functions on the node inputs in the Population Protocol model [2], in the presence of crash faults and transient faults that can corrupt the states of the nodes. They construct a transformation which makes tolerant in the presence of such failures any protocol that works in the failure-free setting, as long as modifying a small number of inputs does not change the output. Guerraoui and Ruppert [11] introduced a new model, called *Community Protocol*, which is inspired by the Population Protocol model, but the nodes have unique identifiers and enough memory to store a constant number of other agents' identifiers. They show that this model can solve any decision problem in  $\text{NSPACE}(n \log n)$  while tolerating a constant number of Byzantine failures.

In [13], Peleg studies logical structures, constructed over static graphs, that need to satisfy the same property on the resulting structure after node or edge failures. He distinguishes between the stronger type of fault-tolerance obtained for geometric graphs (termed rigid fault-tolerance) and the more flexible type required for handling general graphs (termed competitive fault-tolerance). It differs from our work, as we address the problem of constructing such structures over dynamic graphs and we study fault-tolerance of distributed models.

*Our contribution:* A Network Constructor (NET) protocol stabilizes to a network, satisfying some graph property  $P$ , starting from an initial configuration where all nodes are in the same state and all connections are disabled. The protocols in [1] do not consider any type of faults, and it is not clear whether they can tolerate even a single fault. In this work, we formally define the model

that extends NET with crash failures, and we examine NET protocols in the presence of such faults. Whenever a node crashes, it is removed from the population, along with all its activated edges. This leaves the remaining population in a state where some actions may need to take place in order to eventually stabilize to a correct network. We answer the following questions: Can we always re-stabilize to a correct graph in this setting, and if not, what is the class of graph properties for which we can always find a fault-tolerant protocol? What are the additional minimal assumptions that we need to make in order to find fault-tolerant protocols for a bigger class of properties?

In Section 3, we study the class of properties for which we are able to design protocols that tolerate any number of faults. We show that this class is non-empty but very small, and then we show that for a wider class of properties, such protocols do not exist, if we do not make further assumptions (e.g. fault notifications or non-constant memory).

The main source of difficulty in the standard NET model (call it SNET) is that after a crash fault, it is not possible for the remaining population to detect the absence of the crashed node, with the purpose of taking actions and eventually re-stabilizing to a correct graph. Also, alive nodes cannot sense the changes in the links that were attached to them before faults occurred (crash faults change the degrees of alive nodes). This means that even if the faults occur only after stabilization, we show for some graph properties that the protocol cannot update the network (in order to fix it), unless it would incorrectly update a stable network in some other execution.

In light of the impossibilities in the SNET model, we introduce the minimal additional assumption of *fault notifications* on some nodes of the population (N-NET model). In particular, after a fault on some node  $u$  occurs, its adjacent nodes (if any) are notified. If no adjacent nodes exist, an arbitrary node in the population is being notified. In that way, we guarantee that at least one node in the population will sense the removal of  $u$ <sup>1</sup>.

In Section 4.1, we give protocols for some otherwise infeasible graph properties that we are now able to construct while tolerating any number of crash failures.

We go one step further, trying to provide universal constructors that can tolerate crash failures. To this end, we allow the nodes to toss an unbiased fair coin during an interaction (PN-NET model), and in Section 4.2 we investigate the more generic question of what is in principle constructible. We call *useful space* the number of nodes that eventually form the graph that satisfies the required property, and *waste* the rest of the population. The idea is based on [1], where they show several universality results by constructing (on  $k$  nodes) of the population a network  $G_1$  capable of simulating a Turing Machine (waste), and then repeatedly construct a random network  $G_2$  on the remaining  $n - k$  nodes (useful space). The idea is to execute on  $G_1$  the Turing Machine which decides the language  $L$  with input the network  $G_2$ . If the Turing Machine accepts, the TM outputs  $G_2$ , otherwise the TM constructs again a random graph. A fault tolerant extension of this is the core idea of our universality results for the PN-NET model, tolerating any number of crash failures.

In order to give fault-tolerant protocols without waste, in Section 4.3 we design a protocol that can be composed in parallel with any N-NET protocol in order to make it fault-tolerant. The idea is to restart the protocol whenever a crash failure occurs. We show that restarting is impossible with constant local memory, if the nodes form unbounded number of connections. To this end, we need to supply the agents with more memory (at most logarithmic on the population size).

Finally, in Section 5 we conclude and discuss further interesting open problems.

---

<sup>1</sup>Some constructions work without notifications in the case of a crash failure on an isolated node, but for some of them it is essential.

## 2 Model and Definitions

A Standard Network Constructor (*SNET*) is a distributed protocol defined by a 4-tuple  $(Q, q_0, Q_{out}, \delta)$ , where  $Q$  is a finite set of node-states,  $q_0 \in Q$  is the initial node-state,  $Q_{out} \subseteq Q$  is the set of output node-states, and  $\delta : Q \times Q \times \{0, 1\} \rightarrow Q \times Q \times \{0, 1\}$  is the transition function. The system consists of a population  $V_I$  of  $n$  distributed *processes* (also called *nodes*). In the generic case, there is an underlying *interaction graph*  $G_I = (V_I, E_I)$  specifying the permissible interactions between the nodes. In this work,  $G_I$  is a *complete undirected interaction graph*, i.e.  $E_I = \{uv : u, v \in V_I \text{ and } u \neq v\}$ .

The main difference between this model and the Population Protocol model is that the edges have binary states (*active* or *inactive*). In other words, we say that the nodes are allowed to form connections between them. During a (pairwise) interaction, the agents are allowed to access the state of their joining edge and either activate it (*state* = 1) or deactivate it (*state* = 0). When the edge state between two nodes  $u$  and  $v$  is activated, we say that  $u$  and  $v$  are *connected*, or *adjacent* at that time  $t$ , and we write  $u \sim_t v$ . Initially, all nodes are in the same state  $q_0$  and all connections are inactive. The goal is for the processes, after interacting and activating/deactivating connections for a while, to end up with a desired stable network, which satisfies some graph property  $P$ .

In this work, we present a version of this model that allows *adversarial crash failures*. A crash (or *halting*) failure causes an agent to cease functioning and play no further role in the execution. We also discuss about *edge failures* throughout the paper. An edge failure disconnects two adjacent nodes (i.e. the edge state between two nodes is altered from 1 to 0).

The execution of a protocol proceeds in discrete steps. In every step, a pair of nodes  $uv$  from  $E_I$  is selected by an *adversary scheduler*, subject to some *fairness* guarantee. These nodes interact and update their states and the state of the edge between them according to a joint transition function  $\delta$ . If two agents in states  $q_u$  and  $q_v$  with the edge joining them in state  $q_{uv}$  encounter each other, they can change into states  $q'_u$ ,  $q'_v$  and  $q'_{uv}$ , where  $(q'_u, q'_v, q'_{uv}) \in \delta(q_u, q_v, q_{uv})$ . Without loss of generality, assume that the transition function is *symmetric*:  $\delta(q_u, q_v, q_{uv}) = \delta(q_v, q_u, q_{uv})$ .

A configuration is a mapping  $C : V_I \cup E_I \rightarrow Q \cup \{0, 1\}$  specifying the state of each node and each edge of the interaction graph. An execution of the protocol on input  $I$  is a finite or infinite sequence of configurations,  $C_0, C_1, C_2, \dots$ , each of which is a multiset of states drawn from  $Q \cup \{0, 1\}$ . In the initial configuration  $C_0$ , all nodes are in state  $q_0$  and all edges are inactive. A configuration  $C_k$  is obtained from  $C_{k-1}$  by one of the following types of transitions:

1. **Ordinary transition:**  $C_k = (C_{k-1} - \{q_u, q_v, q_{uv}\}) \cup \{q'_u, q'_v, q'_{uv}\}$  where  $\{q_u, q_v, q_{uv}\} \subseteq C_{k-1}$  and  $(q'_u, q'_v, q'_{uv}) \in \delta(q_u, q_v, q_{uv})$ .
2. **Crash failure:**  $C_k = C_{k-1} - \{q_u\} - \{q_{uv} : uv \in E_I\}$  where  $\{q_u, q_{uv}\} \subseteq C_{k-1}$ .
3. **Null step:**  $C_k = C_{k-1}$ .

We say that  $C'$  is *reachable from*  $C$  and write  $C \rightsquigarrow C'$ , if there is a sequence of configurations  $C = C_0, C_1, \dots, C_t = C'$ , such that  $C_i \rightarrow C_{i+1}$  for all  $i$ ,  $0 \leq i < t$ . The fairness condition that we impose on the scheduler is quite simple to state. Essentially, we do not allow the scheduler to avoid a possible step forever. More formally, if  $C$  is a configuration that appears infinitely often in an execution, and  $C \rightarrow C'$ , then  $C'$  must also appear infinitely often in the execution. Equivalently, we require that any configuration that is always reachable is eventually reached.

We define the output of a configuration  $C$  as the graph  $G(C) = (V, E)$  where  $V = \{u \in V_I : C(u) \in Q_{out}\}$  and  $E = \{uv : u, v \in V, u \neq v, \text{ and } C(uv) = 1\}$ . If there exists some step  $t \geq 0$  such that  $G(C_i) = G$  for all  $i \geq t$ , we say that the output of an execution  $C_0, C_1, \dots$  *stabilizes* (or

converges) to graph  $G$ , every configuration  $C_i$ , for  $i \geq t$ , is called *output-stable*, and  $t$  is called the *running time* under our scheduler.

Finally, we say that an SNET protocol  $\Pi$  stabilizes eventually to a graph  $G(\Pi)$  of *type*  $P$  if and only if after a finite number of pairwise interactions, the graph defined by 'on' edges does not change and has property  $P$ . We call that stable graph the graph  $G(\Pi)$ .

**Definition 1.** Let  $P$  be a property of graphs. Two graphs  $G$  and  $G'$  are said to be equivalent under property  $P$ , or belong to the same class under  $P$ , if and only if both have property  $P$ . We denote this by  $G \sim_P G'$ .

**Definition 2.** Let  $\Pi$  be an SNET protocol that stabilizes to the graph  $G(\Pi)$ , having property  $P$ .  $\Pi$  is called *k-fault-tolerant* iff there exists a size  $n_0 \geq k$  such that for any population size  $n > n_0$ ,  $\Pi$  stabilizes to a graph  $G' \sim_P G$ , even if a sequence of up to  $k$  crash failures occur during an execution. We also call  $\Pi$  *fault-tolerant* if it stabilizes to a graph  $G' \sim_P G$ , regardless of the number of faults.

To define *N-NETs*, we now extent the Standard Network Constructors model with a *fault flag* in each agent. When a node  $u$  crashes at time  $t$ , every node  $v$  which was adjacent to  $u$  at time  $t$  ( $u \sim_t v$ ) is notified, that is, the fault flag of all  $v$  becomes 1. In the case where  $u$  is an isolated node (i.e. it has no enabled connections), a (random) node  $w$  in the network is notified, and its fault flag becomes 2. At any time, the agents are allowed to access the fault flag and reset it to zero. We call this model *N-NET*.

More formally, the set of node-states is  $Q \times \{0, 1, 2\}$ , and for clarity in our descriptions and protocols, we define two types of transition functions. The first one determines the state/connection updates of pairwise interactions ( $\delta_1 : Q \times Q \times \{0, 1\} \rightarrow Q \times Q \times \{0, 1\}$ ), while the second transition function determines the state updates after a fault ( $\delta_2 : Q \times \{0, 1, 2\} \rightarrow Q \times \{0, 1, 2\}$ ). The first transition function is triggered after a pairwise interaction, while  $\delta_2$  is triggered right after a fault.

The separation of these transition functions is equivalent to the case where only one transition function exists  $\delta : (Q \times \{0, 1, 2\}) \times (Q \times \{0, 1, 2\}) \times \{0, 1\} \rightarrow (Q \times \{0, 1, 2\}) \times (Q \times \{0, 1, 2\}) \times \{0, 1\}$ . Consider the case where a node  $u$  crashes, notifying a node  $w$  in the population (its fault flag becomes either 1 or 2). Then, in the first case (separate transition functions),  $w$  is instantly allowed to update its state, while in the second case (unified transition functions),  $w$  waits until its next interaction with a node  $v$ , applying the rule of  $\delta_2$  independently of the state and connection of  $v$ . During the same interaction,  $w$  and  $v$  can also update their states and connections based on the corresponding rule of  $\delta_1$ .

Finally, we define *PN-NET* in precisely the same way as *N-NET*, but in extension to the above model, every pair of processes is capable of tossing an unbiased coin during an interaction between them.

### 3 On the existence of Fault-Tolerant SNET Protocols

In this section, we study the existence of fault-tolerant protocols in the SNET model. We say that a protocol  $\Pi$  *constructs* a graph property  $P$  if every execution of  $\Pi$  on a population of agents stabilizes on a graph with property  $P$ . We show that not all properties can be constructed by an SNET protocol under faults, but there is a class of properties that has fault-tolerant SNET protocols for any number of crash failures.

**Definition 3.** Let  $G$  be a graph with property  $P$ . Call  $u$  *critical node* of  $G$  if by removing  $u$  at time  $t$  and all its edges, the resulting network  $G' = G - \{u\} - \{uv : v \sim_t u\}$ , does not satisfy property  $P$ .

(i.e.  $G' \not\sim_P G$ ).

In other words, if there are no critical nodes in  $G$ , then any (induced) subgraph  $G'$  of  $G$  that can be obtained by removing nodes and all their edges (crash failures), also satisfy  $P$ . The properties that satisfy this are known as *hereditary properties* in the literature.

**Definition 4.** A property  $P$  is called *hereditary* if for any graph  $G$  with property  $P$ , every induced subgraph of  $G$  also satisfies  $P$ . In other words,  $G$  has no critical nodes.

Examples of *hereditary properties* are “Bipartite graph”, “Planar graph”, “Forest of trees”, “Clique”, “Set of cliques”, “Maximum node degree  $\leq \Delta$ ” and so on. We call *Hereditary* the class of all hereditary properties.

We now define a subclass of this class of properties, which we call *Preserving Graphs* or  $PG$ .

**Definition 5.** A property  $P$  is called *preserving* if for any graph  $G$  with property  $P$ , every subgraph of  $G$  (not necessarily induced) also satisfies  $P$ .

Examples of *preserving properties* are “Bipartite graph”, “Planar graph”, “Maximum node degree  $\leq \Delta$ ” and so on. We call *Preserving Graphs* or  $PG$  the class of all preserving properties.

**Theorem 1.**  $PG$  is a subclass of *Hereditary*.

*Proof.* Consider a property  $P \in PG$ , and a graph  $G$  of type  $P$ . Then, if we remove any node  $u$  and all its edges, the resulting graph  $G'$  should still have property  $P$ , as  $G'$  is subgraph of  $G$  and  $P \in PG$ . Thus,  $P \in Hereditary$ . Now, consider the property  $P = \{clique\}$ . If we remove a node and all its edges from a  $G$  of type  $P$ , the resulting graph  $G'$  is still a clique of smaller size. However, any subgraph of  $G'$  which consists of all the nodes of  $G$  and  $n(n-1)-1$  edges is not a clique. Thus,  $P \in Hereditary$ , but  $P \notin PG$ .  $\square$

**Theorem 2.** If a protocol  $\Pi$  stabilizes to a graph  $G$  of property  $P \in PG$  and if for all  $t$ ,  $G(C_t)$  is a subgraph of  $G(C_{t+1})$  (i.e.  $\Pi$  does not remove any edges), then  $\Pi$  resists any sequence of single faults.

*Proof.* Since  $P \in PG$ , then for each  $t$ ,  $G(C_t)$  has also  $P$ . But then any fault does not destroy the property at any  $t$ .  $\square$

In other words, for any property  $P$  which is preserving, every protocol that stabilizes to a graph  $G$  of some  $P$ , is not necessary to deal with the failures in order to fix the configuration, as this class of graphs has the interesting property of maintaining  $P$  in every subgraph. Note that protocols for properties in  $PG$ , tolerate both crash and edge failures. *Edge failures* corrupt the state of an edge, that is, an activated edge between two nodes is removed, leaving the two corresponding nodes disconnected.

There are some properties  $P \notin PG$  for which we can still design fault-tolerant protocols, without having to deal with the crash failures. An example of such property is the *Spanning Clique*. Let *Clique* be the following 2-state *symmetric* protocol. If we consider the case where no crash faults are allowed, for any population size, *Clique* Protocol stabilizes to a clique with all the nodes in state  $r$  (i.e.  $G(Clique) = \text{“clique on all nodes”}$  and  $P = \text{“clique”}$ ).

**Lemma 1.** *Clique Protocol is fault-tolerant.*

---

**Protocol 1** Clique

---

$$Q = \{b, r\} \times \{0, 1\}$$

Initial state:  $b$

$\delta :$

$$(b, b, 0) \rightarrow (b, r, 0)$$

$$(b, r, 0) \rightarrow (r, r, 0)$$

$$(r, r, 0) \rightarrow (r, r, 1)$$

\\All transitions that do not appear have no effect.

---

*Proof.* Let  $k < n$  and assume that  $k$  nodes crash during the execution. Call  $S$  the remaining  $n - k$  nodes.

- (a) If all nodes in  $S$  are in state  $b$ , then the remaining nodes shall form a clique (in state  $r$ ).
- (b) If all nodes in  $S$  are in state  $r$ , then again, *Clique Protocol* stabilizes to a clique.
- (c) If  $S$  contains both colors, then the  $r$ -nodes will convert the  $b$ -nodes to  $r$  and again *Clique Protocol* stabilizes to a clique.  $\square$

**Definition 6.** A state  $s$  of an SNET protocol  $\Pi$  is called critical iff its disappearance from the population at some execution point makes  $\Pi$  impossible to stabilize to a graph  $G$  of property  $P$  with no crash faults.

This means that if at some point during an execution the population remaining does not have state  $s$  in any node, then  $\Pi$  will either not stabilize to any graph or stabilize to a graph  $G'$  where  $G' \not\sim_P G$ . The following observation holds by the definition of the critical states.

**Observation 1.** An SNET  $\Pi$  is fault-tolerant iff  $\Pi$  has no critical states.

**Theorem 3.** There exists a 2-state SNET protocol  $\Pi$  with at least one critical state. In other words, not all 2-state SNET  $\Pi$  are fault-tolerant.

*Proof.* Let  $\Pi^*$  be the Protocol 2 which constructs a spanning star.

---

**Protocol 2** Spanning Star

---

$$Q = \{b, r\} \times \{0, 1\}$$

Initial state:  $b$

$\delta :$

$$(b, b, 0) \rightarrow (b, r, 1)$$

$$(b, r, 0) \rightarrow (b, r, 1)$$

$$(r, r, 1) \rightarrow (r, r, 0)$$

---

If we do not allow crash faults to happen, then  $\Pi^*$  will stabilize to graph  $G(\Pi^*)$  of type  $P$  = "spanning star", where the center is in state  $b$  and the leaves in state  $r$ .

Now, assume that the adversary waits until one  $b$ -node remains (the center) and then removes it (crashes). Now, only  $r$  nodes remain, and with just one fault,  $\Pi$  will converge to a set of independent vertices in state  $r$  (empty graph). Thus, the state  $b$  in protocol  $\Pi^*$  is critical.  $\square$

Here, it is reasonable to ask whether there exists another SNET protocol  $\Pi'$  which is fault-tolerant and stabilizes to a graph  $G$  of property  $P =$  "spanning star". We call this protocol the "self-stabilizing" version of the *Spanning Star* protocol.

**Theorem 4.** *There exists no SNET  $\Pi'$  which would be the self-stabilizing version of the Spanning Star protocol, even with one fault.*

*Proof.* Assume such an SNET protocol  $\Pi'$  exists. Then  $\Pi'$  should stabilize to a spanning star regardless of whether up to  $k$  faults occur or not. Clearly, in any SNET protocol that stabilizes to a spanning star, the eventual state of the center of the star (say  $b'$ ) will be different from any of the states of the other nodes (leaves). This is because under any fair scheduler, nodes meet infinitely often. Then, the eventual states of the leaves of the star should enforce no edges between them. Thus, if  $b'$  was one of the states of the leaves, then no leaf would be connected to the center. Let us run the protocol  $\Pi'$ , until stabilization, under no faults. Let  $S \subseteq Q$  be the set of states of the leaves, after the spanning star is formed. Now, let the adversary wait until this happens and  $S, b'$  appear. Then, the adversary removes node  $b'$  (crash failure). Since,  $\Pi'$  is fault-tolerant, the rules of  $\Pi'$  should recreate the star. This means that the states in  $S$  and the rules should create edges among the former leaves. But then, even when no faults occur, the same rules and the same sequence of interactions should create edges in  $S$  (among the former leaves). This contradicts the assumption that  $S$  is the set of states of the leaves after the star is formed. Thus, no such  $\Pi'$  can exist.  $\square$

**Corollary 1.** *There is at least one SNET protocol  $\Pi$  which cannot have an equivalent fault-tolerant version  $\Pi'$ .*

In a similar way, we show the following lemma.

**Lemma 2.** *There is no 1-fault-tolerant SNET protocol for constructing a spanning line.*

*Proof.* Assume such an SNET protocol  $\Pi$  exists. Then  $\Pi$  should stabilize to a spanning line regardless of whether up to  $k$  faults occur or not. Clearly, in any SNET protocol that stabilizes to a spanning line, the eventual state (or states) of the endpoints of the line (say  $q_e$ ) will be different from any of the states of the other nodes. This is because under any fair scheduler, nodes meet infinitely often. Then, the eventual states of the inner nodes of the line should enforce no more edges between them and other nodes. Thus, if  $q_e$  was one of the states of the inner nodes, then no more nodes could be connected to the line, thus, the protocol would end up with many disjoint lines. Let us run the protocol  $\Pi$ , until stabilization, under no faults. Let  $S \subseteq Q$  be the set of states of the inner nodes, after the spanning line is formed. Now, let the adversary wait until this happens and  $S, q_e$  appear. Then, the adversary removes an inner node (in state  $q \in S$ ) from the line (crash failure). Since,  $\Pi$  is fault-tolerant, the rules of  $\Pi$  should recreate the spanning line. This means that the states in  $S$  and the rules of  $\Pi$  should create edges among the former inner nodes. But then, even in the case where no faults occur, the same rules and the same sequence of interactions should create edges in  $S$ , among the former inner nodes (i.e. a cycle is formed). This contradicts the assumption that  $S$  is the set of states of the inner nodes after the spanning line is formed. Thus, no such  $\Pi$  can exist.  $\square$

We now show that if there exists at least one critical node in  $G$ , there is no SNET protocol that always stabilizes to the correct network even if a single failure occurs during an execution.

**Theorem 5.** *If there exists a critical node in  $G$ , there is no 1-fault tolerant SNET protocol that stabilizes to it.*



*Proof.* Let  $\Pi$  be an SNET protocol that stabilizes to graph  $G$ , having property  $P$  and tolerating one crash failure. Consider an execution  $E$  and a sequence of configurations  $C_0, C_1, \dots$  of  $E$ . Assume a time  $t$  that the output of  $E$  has stabilized to graph  $G$  (i.e.  $G(C_i) = G, \forall i \geq t$ ). Let  $u$  be a critical node in  $G$ . Assume that the scheduler removes  $u$  and all its edges (crash failure) at time  $t' > t$ , resulting to a graph  $G' \not\sim_P G$ . In order to fix the network, the protocol must change at some point  $t''$  the configuration, for example a node  $v$  changes its state. Now, call  $E'$  the execution that node  $u$  does not crash, and between  $t'$  and  $t''$  the node  $v$  has the same interactions as in the previous case where node  $u$  crashed. Then,  $v$  changes its state in order to fix the network, since it cannot distinguish  $E$  from  $E'$ . The fact that  $u$  either crashes or not, leads to the same result (i.e.  $v$  tries to fix the network thinking that  $u$  has crashed). This means that if we are constantly trying to detect faults in order to deal with them, this would happen indefinitely and the protocol would never be stabilizing. Consider that the network has stabilized to  $G$ . At some point, because of the infinite execution, a node will surely but wrongly detect a crash failure. Thus,  $G$  has not really stabilized.  $\square$

## 4 Notified Network Constructors

In this section, we use the N-NET model as described in Section 2, and we investigate whether the additional information in each agent (the fault flag) is sufficient in order to design fault-tolerant or  $k$ -fault-tolerant protocols, overcoming the impossibility of certain graph properties in the SNET model (graphs with critical nodes).

### 4.1 Fault-tolerant N-NET protocols via minimal updates

In this section, our goal is to design protocols that after a fault, the nodes try to fix the configuration with minimal updates and eventually stabilize to a correct network. We give protocols for some properties, such as *spanning star*, *cycle cover*, and in Section 4.2 we give a fault-tolerant spanning line protocol which is part of our generic constructor capable of constructing a large class of networks.

---

#### Protocol 3 FT Spanning Star

---

$$Q = \{b, r\} \times \{0, 1\}$$

Initial state:  $b$

$\delta_1 :$

$$(b, b, 0) \rightarrow (b, r, 1)$$

$$(b, b, 1) \rightarrow (b, r, 1)$$

$$(r, r, 1) \rightarrow (b, b, 0)$$

$$(b, r, 0) \rightarrow (b, r, 1)$$

$\delta_2 :$

$$(r, 1) \rightarrow (b, 0)$$


---

**Lemma 3.** *FT Spanning Star is fault-tolerant.*

*Proof.* Assume that any number of faults  $k < n$  occur during an execution. Initially, all nodes are in state  $b$  (*black*). Two nodes connect with each other, if either one of them is black, or both of them are black, in which case one of them becomes  $r$  (*red*). A black node can become red only

by interaction with another black node, in which case they also become connected. Thus, with no crash faults, a connected component always includes at least one black node. In addition, all isolated nodes are always in state  $b$ . This is because, if a red node removes an edge it becomes black.

Then, if a (connected) node crashes, the adjacent nodes are notified and the red nodes become black, thus, any connected component should again include at least one black node. Now, consider the case where only one black node remains in the population. Then the rest of the population (in state  $r$ ) should be in the same connected component as the unique  $b$  node. Then, if  $b$  crashes, at least one black node will appear, thus, this protocol maintains the invariant, as there is always at least one black node in the population. *FT Spanning Star* then stabilizes to a star with a unique black node in the center.  $\square$

---

**Protocol 4** FT Cycle-Cover

---

$Q = \{q_0, q_1, q_2\} \times \{0, 1\}$

Initial state:  $q_0$

$\delta_1 :$

$(q_0, 0) \rightarrow (q_1, 1)$

$(q_1, 0) \rightarrow (q_2, 1)$

$(q_1, 1) \rightarrow (q_2, 0)$

$\delta_2 :$

$(q_1, 1) \rightarrow (q_0, 0)$

$(q_2, 1) \rightarrow (q_1, 0)$

---

Similarly, we can show the following lemma.

**Lemma 4.** *FT Cycle-Cover is fault-tolerant.*

## 4.2 Universal Fault-Tolerant Constructors with waste

In this section, we ask whether there is a generic fault-tolerant constructor capable of constructing a large class of networks. We first give a fault-tolerant protocol that constructs a spanning line, and then we show that we can simulate a given TM on that line, tolerating any number of crash faults.

**Lemma 5.** *FT Spanning Line is fault-tolerant.*

*Proof.* Initially, all nodes are in state  $q_0$  and they start connecting with each other in order to form lines that eventually merge into one.

When two  $q_0$  nodes become connected, one of them becomes leader (state  $l_0$ ) and starts connecting with  $q_0$  nodes (expands). A leader state  $l_0$  is always an endpoint. The other endpoint is in state  $e_i$  (initially  $e_1$ ), while the inner nodes are in state  $q_2$ . Our goal is to have only one leader  $l_0$  on one endpoint, because  $l_0$  are also used in order to merge lines. Otherwise, if there are two  $l_0$  endpoints, the line could form a cycle.

When two  $l_0$  leaders meet, they connect (line merge) and a  $w$  node appears. This state performs a random walk on the line and its purpose is to meet both endpoints (at least once) before becoming an  $l_0$  leader. After interacting with the first endpoint, it becomes  $w_1$  and changes the endpoint to

$e_1$ . Whenever it interacts with the same endpoint they just swap their states from  $e_1, w_1$  to  $e_2, w_2$  and vice versa. In this way, we guarantee that  $w_i$  will eventually meet the other endpoint in state  $e_j$ ,  $j \neq i$ , or  $l_0$ . In the first case, the  $w_i$  node becomes a leader ( $l_0$ ), after having walked the whole line at least once.

Now, consider the case where a fault may happen on some node on the line. If the fault flag of an endpoint state becomes 1, it updates its state to  $q_0$ . Otherwise, the line splits into two disjoint lines and the new endpoints become  $l_1$ . An  $l_1$  becomes a walking state  $w_1$ , changes the endpoint into  $e_1$  and performs the same process (random walk).

If there are more than one walking states on a line, then all of them are  $w$ , or  $w_i$  and they perform a random walk. None of them can ever satisfy the criterion to become  $l_0$  before first eliminating all the other walking states and/or the unique leader  $l_0$  (when two walking states meet, only one survives and becomes  $w$ ), simply because they form natural obstacles between itself and the other endpoint. If a new fault occurs, then this can only introduce another  $w_i$  state which cannot interfere with what existing  $w_i$ 's are doing on the rest of the line (can meet them eventually but cannot lead them into an incorrect decision).

If an  $l_0$  leader is merging while there are  $w_i$ 's and/or  $w$ 's on its line (but it is not aware of that), the merging results in a new  $w$  state, which is safe because a  $w$  cannot make any further progress without first succeeding to beat everybody on the line. A  $w$  can become  $l_0$  only after walking the whole line at least once (i.e. interact with both endpoints) and to do that it must have managed to eliminate all other walking states of the line on its way.  $\square$

---

**Protocol 5** FT Spanning Line

---

 $Q = \{q_0, q_2, e_1, e_2, l_0, l_1, w, w_1, w_2\} \times \{0, 1\}$ Initial state:  $q_0$  $\delta_1 :$  $(q_0, q_0, 0) \rightarrow (e_1, l_0, 1)$  $(l, q_0, 0) \rightarrow (q_2, l_0, 1)$  $(l_0, l_0, 0) \rightarrow (q_2, w, 1)$  $\backslash \backslash w$  nodes perform a random walk on line $(l_1, q_2, 1) \rightarrow (e_1, w_1, 1)$  $(w_i, q_2, 1) \rightarrow (q_2, w_i, 1)$  $(w, q_2, 1) \rightarrow (q_2, w, 1)$  $(w, e_i, 1) \rightarrow (w_i, e_i, 1)$  $(w_i, e_i, 1) \rightarrow (w_j, e_j, 1), i \neq j$  $(w_i, e_j, 1) \rightarrow (q_2, l_0, 1), i \neq j$  $(w, l_i, 1) \rightarrow (w_1, e_1, 1)$  $(w_i, l_i, 1) \rightarrow (q_2, l_0, 1)$  $\backslash \backslash w$  nodes eliminate each other, until only one survives $(w_i, w_j, 1) \rightarrow (w, q_2, 1)$  $(w, w_j, 1) \rightarrow (w, q_2, 1)$  $\delta_2 :$  $(e_1, 1) \rightarrow (q_0, 0)$  $(e_2, 1) \rightarrow (q_0, 0)$  $(l_0, 1) \rightarrow (q_0, 0)$  $(l_1, 1) \rightarrow (q_0, 0)$  $(q_2, 1) \rightarrow (l_1, 0)$  $(w, 1) \rightarrow (l_1, 0)$  $(w_1, 1) \rightarrow (l_1, 0)$  $(w_2, 1) \rightarrow (l_1, 0)$ 

---

**Lemma 6.** *There is an N-NET  $\Pi$  such that when  $\Pi$  is executed on  $n$  nodes and at most  $k$  faults can occur,  $0 \leq k < n$ ,  $\Pi$  will eventually simulate a given TM  $M$  of space  $O(n-k)$  in a fault-tolerant way.*

*Proof.* The state of  $\Pi$  has two components  $(P, S)$ , where  $P$  is executing a spanning line formation procedure, while  $S$  handles the simulation of the TM  $M$ . Our goal is to eventually construct a spanning line, where initially the state of the second component of each node is in an initial state  $s_0$  except from one node which is in state *head* and indicates the head of the TM.

In general, the states  $P$  and  $S$  are updated in parallel and independently from each other, apart from some cases where we may need to reset either  $P$ ,  $S$  or both.

In order to form a spanning line under crash failures, the  $P$  component will be executing our *FT Spanning Line* protocol which is guaranteed to construct a line, spanning eventually the non-faulty nodes.

It is sufficient to show that the protocol can successfully reinitialize the state of all nodes on the line after a final *event* has happened and the line is stable and spanning. Such an *event* can be a line merging, a line expansion, a fault on an endpoint or an intermediate fault. The latter though can only be a final event if one of the two resulting lines is completely eliminated due to faults before merging again. In order to re-initialize the TM when the line expands to an isolated node  $q_0$ , we alter a rule of the *FT Spanning Line* protocol. Whenever, a leader  $l_0$  expands to an isolated node  $q_0$ , the leader becomes  $q_2$  while the node in  $q_0$  becomes  $l_1$ , thus introducing a new walking state.

We now exploit the fact that in all these cases, *FT Spanning Line* will generate a  $w$  or a  $w_i$  state in each affected component.

Whenever a  $w_1$  or  $w_2$  state has just appeared or interacted with an endpoint  $e_1$  or  $e_2$  respectively, it starts resetting the simulation component  $S$  of every node that it encounters. If it ever manages to become a leader  $l_0$ , then it finally restarts the simulation on the  $S$  component by reintroducing to it the *tape head*.

When the last event occurs, the final spanning line has a  $w$  or  $w_i$  leader in it, and we can guarantee a successful restart due to the following invariant. Whenever a line has at least one  $w/w_i$  state and no further events can happen, *FT Spanning Line* guarantees that there is one  $w$  or  $w_i$  that will dominate every other  $w/w_i$  state on the line and become an  $l_0$ , while having traversed the line from endpoint to endpoint at least once.

In its final departure from one endpoint to the other, it will dominate all  $w$  and  $w_i$  states that it will encounter (if any) and reach the other endpoint. Therefore, no other  $w/w_i$  states can affect the simulation components that it has reset on its way, and upon reaching the other endpoint it will successfully introduce a new *head* of the TM while all simulation components are in an initial state  $s_0$ .  $\square$

**Lemma 7.** *There is a fault-tolerant N-NET protocol  $\Pi$  which partitions the nodes into two groups  $U$  and  $D$  with waste at most  $2f(n)$ , where  $f(n)$  is an upper bound on the number of faults that can occur.  $U$  is a spanning line with a unique leader in one endpoint and can eventually simulate a TM  $M$ . In addition, each node of  $D$  is connected with exactly one node of  $U$ , and vice versa.*

*Proof.* Initially all nodes are in state  $q_0$ . Protocol  $\Pi$  partitions the nodes into two equal sets  $U$  and  $D$  and every node maintains its type forever. This is done by a perfect matching between  $q_0$ 's where one becomes  $q_u$  and the other becomes  $q_d$ . Then, the nodes of  $U$  execute the *FT Spanning Line* protocol, which guarantees the construction of a spanning line, capable of simulating a TM (Lemma 6). The rest of the nodes ( $D$ ), which are connected to exactly one node of  $U$  each, are used to construct on them random graphs. Whenever a line merges with another line or expands towards an isolated node, the simulation component  $S$  in the states of the line nodes, as described in Lemma 6, is reinitialised sequentially.

Assume that a fault occurs on some node of the perfect matching before that pair has been attached to a line. In this case, it's pair will become isolated therefore it is sufficient to switch that back to  $q_0$ .

If a fault occurs on a  $D$  node  $u$  after its pair  $w$  has been attached to a line,  $w$  goes into a detaching state which disconnects it from its line neighbors, turning them into  $l_1$  and itself becoming a  $q_0$  upon release. An  $l_1$  state on one endpoint is guaranteed to walk the whole line at least once (as  $w_i$ ) in order to ensure that a unique leader  $l_0$  will be created. If  $u$  fails before completing this process, it's neighbors on the line shall be notified becoming again  $l_1$ , and if one of its neighbors fails we shall treat this as part of the next type of faults. This procedure shall disconnect the line but may leave the component connected through active connections within  $D$ . But this is fine as long as the

*FT-Spanning Line* guarantees a correct restart of the simulation after any event on a line. This is because eventually the line in  $U$  will be spanning and the last event will cause a final restart of the simulation on that line.

Assume that a fault occurs on a node  $u \in U$  that is part of the line. In this case the neighbors of  $u$  on the line shall instantly become  $l_1$ . Now, its  $D$  pair  $v$ , which may have an unbounded number of  $D$  neighbors at that point, becomes a special *deactivating state* that eventually deactivates all connections and never participates again in the protocol, thus, it stays forever as waste. This is because the fault partially destroys the data of the simulation, thus, we cannot safely assume that we can retrieve the degree of  $v$  and successfully deactivate all edges. As there can be at most  $f(n)$  such faults we have an additional waste of  $f(n)$ . Now, consider the case where  $u$  is one neighbor of a node  $w$  which is trying to release itself after its  $v$  neighbor in  $D$  failed. Then,  $w$  implements a 2-counter in order to remember how many of its alive neighbours have been deactivated by itself or due to faults in order to know when it should become  $q_0$ .  $\square$

**Theorem 6.** *For any graph language  $L$  that can be decided by a linear space TM, there is a fault tolerant PN-NET  $\Pi$  that constructs a graph in  $L$  with waste at most  $\min\{n/2 + f(n), n\}$ , where  $f(n)$  is an upper bound on the number of faults that can occur.*

*Proof.* By Lemma 7, there is a protocol that constructs two groups  $U$  and  $D$  of equal size, where each node of  $U$  is matched with exactly one node of  $D$ , and vice versa. In addition, the nodes of  $U$  form a spanning line, and by Lemma 6 it can simulate a TM  $M$ . After the last fault occurs,  $M$  is correctly initialized and the head of the TM is on one of the endpoints of the line. The two endpoints are in different states, and assume, that the endpoint that the head ends up is in state  $e_l$  (*left endpoint*), and the other is in state  $e_r$  (*right endpoint*).

We now provide the protocol that performs the simulation of the TM  $M$ , which we separate into several subroutines. The first subroutine is responsible for simulating the direction on the tape and is executed once the head reaches the endpoint  $e_l$ . The simulation component  $S$  (as in Lemma 6) of each node has three sub-components  $(h, c, d)$ .  $h$  is used to store the head of the TM, i.e. the actual state of the control of the TM,  $c$  is used to store the symbol written on each cell of the TM, and  $d$  is either  $l$ ,  $r$  or  $\sqcup$ , indicating whether that node is on the left or on the right of the head (or unknown). Assume that after the initialization of the TM,  $d = \sqcup$  for all nodes of the line. Finally, whenever the head of the TM needs to move from a node  $u$  to a node  $w$ ,  $h_w \leftarrow h_u$ , and  $h_u \leftarrow \sqcup$ .

*Direction.* Once the head of the TM is introduced in the endpoint  $e_l$  by the lines' leader, it moves on the line, leaving  $l$  marks on the  $d$  component of each node. It moves on the nodes which are not marked, until it eventually reaches the  $e_r$  endpoint. At that point, it starts moving on the marked nodes, leaving  $r$  marks on its way back. Eventually, it reaches again the  $e_l$  endpoint. At that time, for each node on its right it holds that  $d = r$ . Now, every time it wants to move to the right it moves onto the neighbor that is marked by  $r$  while leaving an  $l$  mark on its previous position, and vice versa. Once the head completes this procedure, it is ready to begin working as a TM.

*Constructing a random graph in  $D$ .* This subroutine of the protocol constructs a random graph in the nodes of  $D$ . In the *Probabilistic N-NET* model, the nodes are allowed to toss a fair coin during an interaction. This means that we allow transitions that with probability  $1/2$  give one outcome and with  $1/2$  another. To achieve the construction of a random graph, the TM implements a binary counter  $C$  ( $\log n$  bits) in its memory and uses it in order to uniquely identify the nodes of set  $D$  according to their distance from  $e_l$ . Whenever it wants to modify the state of edge  $(i, j)$  of the network in  $D$ , the head assigns special marks to the nodes in  $D$  at distances  $i$  and  $j$  from the left of the endpoint  $e_l$ . Note that the TM uses its (distributed) binary counter in order to count these

distances. If the TM wants to access the  $i$ -th node in  $D$ , it sets the counter  $C$  to  $i$ , places a mark on the left endpoint  $e_l$  and repeatedly moves the mark one position to the right, decreasing the counter by one in each step, until  $C = 0$ . Then, the mark has been moved exactly  $i$  positions to the right. In order to construct a random graph in  $D$ , it first assigns a mark  $r_1$  to the first node  $e_l$ , which indicates that this node should perform random coin tosses in its next interactions with the other marked nodes, in order to decide whether to form connections with them, or not. Then, the leader moves to the next node on its line and waits to interact with the connected node in  $D$ . It assigns a mark  $r_2$ , and waits until this mark is deleted. The two nodes that have been marked ( $r_1$  and  $r_2$ ), will eventually interact with each other, and they will perform the (random) experiment. Finally the second node deletes its mark ( $r_2$ ). The head then, moves to the next node and it performs the same procedure, until it reaches the other endpoint  $e_r$ . Finally, it moves back to the first node (marked as  $r_1$ ), deletes the mark and moves one step right. This procedure is repeated until the node that should be marked as  $r_1$  is the right endpoint  $e_r$ . It does not mark it and it moves back to  $e_l$ . The result is an equiprobable construction of a random graph. In particular, all possible graphs over  $|D|$  nodes have the same probability to occur. Now, the input to the TM  $M$  is the random graph that has been drawn on  $D$ , which provides an encoding equivalent to an adjacency matrix. Once this procedure is completed, the protocol starts the simulation of the TM  $M$ . There are  $m = \binom{k}{2}$  edges, where  $k = |D|$  and  $M$  has available  $\frac{k}{2} = \sqrt{m}$  space, which is sufficient for the simulation on a  $\sqrt{m}$ -space TM.

*Read edges of  $D$ .* We now present a mechanism, which can be used by the TM in order to read the state of an edge joining two nodes in  $D$ . Note that a node in  $D$  can be uniquely identified by its distance from the endpoint  $e_l$ . Whenever the TM needs to read the edge joining the nodes  $i$  and  $j$ , it sets the counter  $C$  to  $i$ . Assume w.l.o.g. that  $i < j$ . It performs the same procedure as described in the subroutine which draws the random graph in  $D$ . It moves a special mark to the right, decreasing  $C$  by one in each step, until it becomes zero. Then, it assigns a mark  $r_3$  on the  $i$ -th node of  $D$ , and then performs the same for  $C = j$ , where it also assigns a mark  $r_4$  (to the  $j$ -th node). When the two marked nodes ( $r_3$  and  $r_4$ ) interact with each other, the node which is marked as  $r_4$  copies the state of the edge joining them to a flag  $f$  (either 0 or 1), and they both delete their marks. The head waits until it interacts again with the second node, and if the mark has been deleted, it reads the value of the flag  $f$ .

After a simulation, the TM either accepts or rejects. In the first case, the constructed graph belongs to  $L$  and the Turing Machine halts. Otherwise, the random graph does not belong to  $L$ , thus the protocol repeats the random experiment. It constructs again a random graph, and starts over the simulation on the new input.

A final point that we should make clear is that if during the simulation of the TM an event occurs (crash fault, line expansion, or line merging), by Lemma 6 and Lemma 7, the protocol reconstructs a valid partition between  $U$  and  $D$ , the TM is re-initialized correctly, and a unique head is introduced in one endpoint. At that time, edges in  $D$  may exist, but this fact does not interfere with the (new) simulation of the TM, as a new random experiment takes place for each pair of nodes in  $D$  prior to each simulation.  $\square$

We now show that if the constructed network is required to occupy  $1/3$  instead of half of the nodes, then the available space of the TM-constructor dramatically increases from  $O(n)$  to  $O(n^2)$ . We provide a protocol which partitions the population into three sets  $U$ ,  $D$  and  $M$  of equal size  $k = n/3$ . The idea is to use the set  $M$  as a  $\Theta(n^2)$  binary memory for the TM, where the information is stored in the  $k(k-1)/2$  edges of  $M$ .

---

**Protocol 6** 3-Partition

---

 $Q = \{q_0, q_d, q_u, q'_u, q_m, q'_m, q_w, q'_w, s\} \times \{0, 1\}$ Initial state:  $q_0$  $\delta_1 :$ 
$$\begin{aligned} (q_0, 0) &\rightarrow (q'_u, 1) \\ (q'_u, 0) &\rightarrow (q_u, 1) \\ (q'_u, 1) &\rightarrow (q_u, q'_m, 1) \\ (q'_m, 1) &\rightarrow (q_m, 0) \\ (q_w, 1) &\rightarrow (q_0, s, 0) \\ (q_w, q_u, 1) &\rightarrow (q_m, q_u, 1) \\ (q'_w, 1) &\rightarrow (q'_0, s, 0) \\ (q'_w, q_m, 1) &\rightarrow (q'_0, s, 0) \\ (q'_w, q'_m, 1) &\rightarrow (q'_0, q'_u, 0) \\ (s, \cdot, 1) &\rightarrow (s, \cdot, 0) \end{aligned}$$
 $\delta_2 :$ 
$$\begin{aligned} (q'_u, 1) &\rightarrow (q_0, 0) \\ (q_d, 1) &\rightarrow (s, 0) \\ (q_m, 1) &\rightarrow (s, 0) \\ (q_w, 1) &\rightarrow (q_0, 0) \\ (q'_w, 1) &\rightarrow (q'_0, 0) \\ (q'_m, 1) &\rightarrow (q_w, 0) \\ (q_u, 1) &\rightarrow (q'_w, 0) \end{aligned}$$

---

**Lemma 8.** *Protocol 3-Partition partitions the nodes into three groups  $U$ ,  $D$  and  $M$ , with waste  $3f(n)$ , where  $f(n)$  is an upper bound on the number of faults that can occur.  $U$  is a spanning line with a unique leader in one endpoint and can eventually simulate a TM, each node in  $D \cup M$  is connected with exactly one node of  $U$ , and each node of  $U$  is connected to exactly one node in  $D$  and one node in  $M$ .*

*Proof.* Protocol 3-Partition constructs lines of three nodes each, where one endpoint is in state  $q_d$ , the other endpoint in state  $q_m$ , and the center is in state  $q_u$ . The nodes of  $U$  operate as in Lemma 7 (i.e. they execute the *FT Spanning Line* protocol). A (connected) pair of nodes waits until a third node is attached to it, and then the center becomes  $q_u$  and starts executing the *FT Spanning Line* protocol. Note that at some point, it is possible that the population may only consists of pairs in states  $q_d$  and  $q'_u$ . For this reason, we allow  $q'_u$  nodes to connect with each other, forming lines of four nodes. One of the  $q'_u$  nodes becomes  $q_u$  and the other becomes  $q'_m$ . A node in  $q'_m$  becomes  $q_m$  only after deactivating its connection with a  $q_d$  node (its previous pair). This results in lines of three nodes each with nodes in states  $q_d$ ,  $q_u$  and  $q_m$ . Then, the  $q_u$  nodes start forming a line, spanning all nodes of  $U$ . In a failure-free setting, the correctness of this protocol follows from Lemma 7. In addition, by Lemma 6, the TM of the line is initialized correctly after the last occurring event (line expansion, line merging, or crash fault).

If we consider crash failures, it is sufficient to show that eventually  $U$  is a spanning line and  $M$  and  $D$  are disjoint. If a node ever becomes  $q_d$  or  $q_m$ , it might form connections with other nodes in  $D$  or  $M$  respectively, because of a TM simulation. A node in  $M$  never forms connections with nodes in  $D$ . After they receive a fault notification, they become the *deactivating state*  $s$ . A node in state  $s$  is disconnected from any other node, thus, it eventually becomes isolated and never



participates in the execution again. We do this because nodes in  $M$  and  $D$  can form unbounded number of connections. The data of the TM have been partially destroyed (because of the crash failure), therefore it is not safe to assume that we can retrieve the degree of them and successfully re-initialize them.

A node  $u$  in state  $q'_m$  (inner node of a line of four nodes), after a fault notification it becomes  $q_w$ . A node in  $q_w$  waits until its next interaction with a connected node  $v$ . If  $v$  is in state  $q_u$ , this means that now a triple has been formed, thus  $u$  becomes  $q_m$ . If  $v$  is in state  $q_d$ , they delete the edge joining them,  $u$  becomes  $q_0$  and  $v$  becomes  $s$  ( $v$  might have formed connections with other nodes in  $D$ ).

A node  $u$  in  $q_u$ , after a fault notification it becomes  $q'_w$  and waits until its next interaction with a connected node  $v$ . At that point,  $v$  can be either  $q_d$ ,  $q'_m$ , or  $q_m$ . In all cases they disconnect from each other and  $u$  becomes  $q'_0$ . The state  $q'_0$  indicates that the node should release itself from the spanning line in  $U$ . This procedure works as described in Lemma 7, thus, after releasing itself from the line, it becomes  $q_0$ . If  $v$  is in state  $q_d$  or  $q_m$ , it becomes  $s$ . If  $v$  is in state  $q'_m$ , it becomes  $q'_u$ , as its (unique) adjacent node can only be in state  $q_d$ .

A node in  $q'_u$  or  $q_w$ , after a fault notification it becomes  $q_0$  and continues participating in the execution again. Finally, a node in state  $q'_w$ , after receiving a fault notification, it becomes  $q'_0$  (a  $q'_w$  is the result of a fault notification in a  $U$ -node).

Note that a node in any state except from  $q_d$  and  $q_m$  can be re-initialized correctly, thus they may participate in the execution again. It is apparent that no node that might have formed unbounded number of connections can participate in the execution again after a crash fault. This guarantees that the connections in  $D$  and  $M$  can be correctly initialized after the final event, and that no node in  $D \cup M$  can be connected with more than one node in  $U$ . In addition, if a  $U$ -node receives a fault notification, it releases itself from the line, thus introducing new walking states in the resulting line(s). By Lemma 6, this guarantees the correct re-initialization of the TM. Finally, a crash failure can lead in deactivating two more nodes, in the worst case. These nodes never participate in the execution again, thus they remain forever as waste. This means that after  $f(n)$  crash failures, the partitioning will be constructed in  $n - 3f(n)$  nodes.  $\square$

**Theorem 7.** *For any graph language  $L$  that can be decided by a  $(O(n^2) + O(n))$ -space TM, there is a protocol that constructs  $L$  equiprobably with waste at most  $\min\{2n/3 + f(n), n\}$ , where  $f(n)$  is an upper bound on the number of faults.*

*Proof.* Protocol 6 partitions the population in three groups  $U$ ,  $D$  and  $M$  and by Lemma 8, it tolerates any number of crash failures, while initializing correctly the TM after the final event (line expansion, line merging, or crash fault). Reading and writing on the edges of  $M$  is performed in precisely the same way as reading/writing the edges of  $D$  (described in Theorem 6). Thus, the Turing Machine has now a  $O(n^2)$ -space binary memory (the edges of  $M$ ) and  $O(n)$ -space on the edges of the spanning line  $U$ . The random graph is constructed on the  $k$  nodes of  $D$  (useful space), where by Lemma 8,  $k = (n - 3f(n))/3 = n/3 - f(n)$  in the worst case.  $\square$

### 4.3 Designing Fault-Tolerant protocols without waste by assuming non-constant memory per node

A very simple, (yet impractical) idea that could tolerate any number  $k < n$  of faults is to restart the protocol each time a node crashes. The implementation of this idea requires the ability of some nodes to detect the removal of a node.

**Definition 7.** Consider any execution  $E_i$  of a finite protocol  $\Pi$ . There exists a finite number of different executions, and for each execution a step  $t_i$  that  $\Pi$  stabilizes. Call  $C_{i,j}$  the  $j$ -th configuration of execution  $E_i$ , where  $j \leq t_i$ . Then, we call maximum reachable degree of  $\Pi$  the value  $d = \max\{\text{Degree}(G(C_{i,j}))\}, \forall i, j$ .

We first show that even in the case where the whole population is notified about a crash failure, global restart is *impossible for protocols with unbounded maximum reachable degree, if the nodes have constant memory*. However, we provide a protocol that restarts the population, but we supply the agents with  $O(\log n)$  bits of memory. In our approach, we use the *N-NET* model, and if a node  $w$  crashes, the set  $N_w$  of the nodes that are notified, has the task to restart the protocol (i.e. to convert the current configuration into an initial one).

Consider a protocol  $\Pi$  with the initial state  $q_0$ . We define as global restart the process which leads all alive nodes to the initial state  $q_0$  without any enabled connections among them and then  $\Pi$  gradually starts again.

**Theorem 8.** Consider a protocol  $\Pi$  with unbounded maximum reachable degree. Then, global restart of  $\Pi$  is impossible for nodes with constant memory, even if every node  $u$  in the population is notified about the crash failure.

*Proof.* Consider a protocol  $\Pi$  with constant number of states  $k$  and unbounded maximum reachable degree, which stabilizes to a graph  $G$  of property  $P$ . Then any degree more than  $k$  cannot be remembered by a node, that is, a state  $q$  cannot indicate the degree of a node.

Assume that at time  $t$  a crash failure occurs and that there are some edges in the graph (call them *spurious edges*).

Protocol  $\Pi$  is allowed to have rules that are triggered by the fault and try to erase those edges (*erasing process*). We assume that all nodes in the population are notified about the crash failure. But, as long as the nodes are not aware of their degree, they do not know when the edge erasing process stops in order to allow the restart. To stop the erasing process is equivalent to counting the remaining edges and wait until the degree reaches zero. After a node deletes an edge it either stays in the same state or updates it in order to remember it. No more than  $k$  such changes can happen, thus it is impossible to delete all edges and restart  $\Pi$  with constant memory.

So, any self-stabilizing protocol will inherit (after restarting gradually) some arbitrary spurious edges. Thus, global restart is impossible.  $\square$

A very interesting related question is to ask whether a protocol  $\Pi$  with unbounded maximum reachable degree can still stabilize to a correct graph after an unsuccessful restart, where some edges exist in the beginning of the execution. This is equivalent to ask whether  $\Pi$  can still stabilize to a correct  $G$ , if we enable arbitrarily some connections prior to the execution.

**Theorem 9.** Consider an *SNET* protocol  $\Pi$  which stabilizes to a graph  $G$  of property  $P$ . Given that all nodes are in an initial state  $q_0$  and assuming an adversary that can initialize arbitrarily any subset of edges among nodes,  $\Pi$  stabilizes to a graph  $G' \not\sim_P G$ .

*Proof.* Assume w.l.o.g. that  $\Pi$  stabilizes to a spanning line. Since the nodes have constant memory (i.e. constant number of states), there exists at least one state  $q_1$  which  $O(n)$  nodes stabilize to. Consider an execution  $E$  where two nodes  $v$  and  $w$  are in the same state  $q_1$  after stabilization at time  $t$ . Consider also a node  $u$  in state  $q_2$  which is adjacent to  $v$  but not to  $w$ , and that  $u$  and  $w$  never interacted with each other until time  $t$ .

Consider now that the adversary initializes the edge between  $u$  and  $w$  to *on*, and we run an execution of  $\Pi$  which is exactly the same as  $E$  ( $u$  and  $w$  won't update their connection state, as they do not interact until  $t' > t$ ). Then, node  $u$  stabilizes having three enabled connections. Since  $v$  and  $w$  are both in the same state  $q_1$ ,  $u$  cannot distinguish  $v$  and  $w$ . If there was a rule in  $\Pi$  which disconnects  $q_2$  and  $q_1$ , this would also happen in the case where  $u$  was not adjacent to  $w$ , resulting  $\Pi$  to stabilize to a graph with at least two disjoint lines, as  $u$  would be disconnected from  $v$ .  $\square$

In light of the impossibility result of Theorem 8, we allow the nodes to use non-constant local memory in order to develop a fault tolerating procedure based on restart. Our goal is to come up with a protocol  $A$  that can be composed with any N-NET protocol  $\Pi$ , so that their composition is a fault-tolerant version of  $\Pi$ . Essentially, whenever a fault occurs,  $A$  will restart all nodes in a way equivalent to as if a new execution of  $\Pi$  had started on the whole remaining population.

We give a protocol that achieves this as follows. All nodes are initially leaders. Through a standard pairwise leader elimination procedure, a unique leader would be guaranteed to remain in the absence of failures. But because a fault can remove the last remaining leader, the protocol handles this by generating a new leader upon getting a fault notification. This guarantees the existence of at least one leader in the population and eventually (after the last fault) of a unique one. There are two main events that trigger a new restarting phase: a fault and a leader elimination. As any new event must trigger a new restarting phase that will not interfere with an outdated one, eventually overriding the latter and restarting all nodes once more, we use phase counters to distinguish among phases. In the presence of a new event it is always guaranteed that a leader at maximum phase will eventually increase its phase, therefore a restart is guaranteed after any event. The restarts essentially cause gradual deactivation of edges (by having nodes remember their degree throughout) and restoration of nodes' states to  $q_0$ , thus executing  $\Pi$  on a fresh initial configuration. For the sake of clarity, we first present a simplified version of the restart protocol that guarantees resetting the state of every node to a uniform initial state  $q_0$ . So, for the time being we may assume that the protocol to be restarted through composition is any Population Protocol  $\Pi$  that always starts from the uniform  $q_0$  initial configuration (all  $u \in V$  in  $q_0$  initially). Later on we shall extend this to handle with protocols that are Network Constructors instead.

**Description of the PP Restarting Protocol.** The state of every node consists of two components  $C_1$  and  $C_2$ .  $C_1$  runs the restart protocol  $A$  while  $C_2$  runs the given PP  $\Pi$ . In general, they run in parallel with the only exception when  $A$  restarts  $\Pi$ . The  $C_1$  component of every node stores a *leader* variable, taking values from  $\{l, f\}$ , and is initially  $l$ , a *phase* variable, taking values from  $\mathbb{N}_{\geq 0}$ , initially 0, and a *fault* binary flag, initially 0.

The transition function is as follows. We denote by  $x(u)$  the value of variable  $x$  of node  $u$  and  $x'(u)$  the value of it after the transition under consideration.

If a leaders' flag becomes 1 or 2, it sets it to 0, increases its phase by one, and restarts  $\Pi$ . If a followers' flag becomes 1 or 2, it sets it to 0, increases its phase by one, becomes a leader, and restarts  $\Pi$ . We now distinguish three types of interactions.

When a leader  $u$  interacts with a leader  $v$ , one of them remains leader (state  $l$ ) and the other becomes a follower (state  $f$ ), both set their phase variable to  $\max\{\text{phase}(u), \text{phase}(v)\} + 1$  and both reset their  $C_2$  component (protocol  $\Pi$ ) to  $q_0$  (i.e. restart  $\Pi$ ).

When a leader  $u$  interacts with a follower  $v$ , if  $\text{phase}(u) = \text{phase}(v)$ , do nothing in  $C_1$  but execute a transition of  $\Pi$  (both  $u$  and  $v$  involved). If  $\text{phase}(u) < \text{phase}(v)$ , then both set their phase variable to  $\max\{\text{phase}(u), \text{phase}(v)\} + 1$  and both restart  $\Pi$ , and finally, if  $\text{phase}(u) > \text{phase}(v)$ , then  $\text{phase}'(v) = \text{phase}(u)$  and  $v$  restarts  $\Pi$ .

When a follower  $u$  interacts with a follower  $v$ , if  $\text{phase}(u) = \text{phase}(v)$  do nothing in  $C_1$  but

execute transition of  $\Pi$ . If  $\text{phase}(u) > \text{phase}(v)$ , then  $v$  sets  $\text{phase}'(v) = \text{phase}(u)$  and  $v$  restarts  $\Pi$ , and finally, if  $\text{phase}(u) < \text{phase}(v)$ , then  $u$  sets  $\text{phase}'(u) = \text{phase}(v)$  and  $u$  restarts  $\Pi$ .

We now show that given any such PP  $\Pi$ , the above restart protocol  $A$  when composed as described with  $\Pi$ , gives a fault-tolerant version of  $\Pi$  (tolerating any number of crash faults).

**Lemma 9** (Leader Election). *In every execution of  $A$ , a configuration  $C$  with a unique leader is reached, such that no subsequent configuration violates this property.*

*Proof.* If after the last fault there is still at least one leader, then from that point on at least one more leader appears (due to the fault flags) and only pairwise eliminations can decrease the number of leaders. But pairwise elimination guarantees eventual stabilization to a unique leader. It remains to show that there must be at least one leader after the last fault. The leader state becomes absent from the population only when a unique leader crashes. This generates a notification, raising at least one follower's fault flag, thus introducing at least one leader.  $\square$

Call a *leader-event* any interaction that changes the number of leaders. Observe that after the last leader-event in an execution there is a stable unique leader  $u_l$ .

**Lemma 10** (Final Restart). *On or after the last leader-event,  $u_l$  will go to a phase such that  $\text{phase}(u_l) > \text{phase}(u)$ ,  $\forall u \in V' \setminus \{u_l\}$ , where  $V'$  denotes the remaining nodes after the crash faults. As soon as this happens for the first time, let  $S$  denote the set of nodes that have restarted  $\Pi$  exactly once on or after that event. Then  $\forall u \in V' \setminus S$ ,  $u \in S$ , an interaction between  $u$  and  $v$  results in  $S \leftarrow S \cup \{u\}$ . Thus,  $S$  will eventually be  $S = V'$ .*

*Proof.* We first show that on or after the last leader-event there will be a configuration in which  $\text{phase}(u_l) > \text{phase}(u)$ ,  $\forall u \in V' \setminus \{u_l\}$  and it is stable. As there is a unique leader  $u_l$  and follower-to-follower interactions do not increase the maximum phase within the followers population,  $u_l$  will eventually interact with a node that is in the maximum phase. At that point it will set its phase to that maximum plus one and we can agree that before that follower also sets its own phase during that interaction to the new max, it has been satisfied that  $\text{phase}(u_l) > \text{phase}(u)$ ,  $\forall u \in V' \setminus \{u_l\}$ .

When the above is first satisfied,  $S = \{u_l, u\}$  and  $\text{phase}(u_l) = \text{phase}(u) > \text{phase}(v)$ ,  $\forall v \in V' \setminus S$ . Any interaction within  $S$ , only executes a normal transition of  $\Pi$ , as in  $S$  they are all in the same phase. Any interaction between a  $u \in V' \setminus S$  and a  $v \in S$ , results in  $S \leftarrow S \cup \{u\}$ , because interactions between followers in  $V' \setminus S$  cannot increase the maximum phase within  $V' \setminus S$ , thus  $\text{phase}(v) > \text{phase}(u)$  holds and the transition is:  $\text{phase}'(u) = \text{phase}(v)$  and  $u$  restarts  $\Pi$ , thus enters  $S$ . It follows that  $S$  cannot decrease and any interaction between the two sets increases  $S$ , thus  $S$  eventually becomes equal to  $V'$ .  $\square$

Putting Lemma 9 and Lemma 10 together gives the aforementioned result.

**Theorem 10.** *For any such PP  $\Pi$ , it holds that  $(A, \Pi)$  is a fault-tolerant version of  $\Pi$ .*

**Lemma 11.** *The required memory in each agent for executing protocol  $A$  is  $O(\log n)$  bits.*

*Proof.* Initially all nodes are potential leaders, and they eliminate each other, moving to next phases at the same time. In the worst case, a single leader  $u$  will eliminate every other leader, turning them into followers, thus in a failure-free setting the phase of  $u$  becomes at most  $n - 1$ . If we consider the case where crash faults may occur, each fault can result in notifying the whole population. This will happen if  $u$  was adjacent to every other node by the time it crashed. Thus, all nodes increase their phase by one and become leaders again. In the worst case, a single leader eliminates

all the other leaders, thus, after the first fault, the maximum phase will be increased by  $n - 2$ . The maximum phase than can be reached is  $\sum_{i=0}^k (n - i) = O(kn)$ , where  $k$  is the maximum number of faults that may occur ( $k < n$ ). Thus, each node is required to have  $O(\log n)$  bits of memory.  $\square$

***N-NET Restarting Protocol.*** We are now extending the *PP Restarting Protocol* in order to handle any N-NET protocol  $\Pi$ . Call this new protocol  $B$ . We store in the  $C_1$  component of each node  $u \in V$  a *degree* variable, that is, whenever a connection is formed or deleted,  $u$  increases or decreases the value of *degree* by one respectively. In addition, whenever the *fault flag* of a node  $u$  becomes one, it means that an adjacent node of it has crashed, thus it decreases *degree* by one. In the case of Network Constructors, the nodes cannot instantly restart the protocol  $\Pi$  by setting their state to the initial one  $q_0$ . By Theorem 9, it is evident that we first need to remove all the edges in order to have a successful restart and eventually stabilize to a correct network.

We now define an intermediate phase, called *Restarting Phase R*, where the nodes that need to be restarted enter by setting the value of a variable *restart* to 1 (stored in the  $C_1$  component). As long as their degree is more than zero, they do not apply the rules of the protocol  $\Pi$  in their second component  $C_2$ , but instead they deactivate their edges one by one. Eventually their degree reaches zero, and then they set *restart* to 0 and continue executing protocol  $\Pi$ . We can say that a node  $u$ , which is in phase  $i$  ( $\text{phase}(u) = i$ ), becomes available for interactions of  $\Pi$  (in  $C_2$ ) only after a successful restart. This guarantees that a node  $u$  will not start executing the protocol  $\Pi$  again, unless its degree firstly reaches zero.

The additional Restarting Phase does not interfere with the execution of the *PP Restarting Protocol*, but it only adds a delay on the stabilization time.

**Lemma 12.** *The variable degree of a node  $u$  always stores its correct degree.*

*Proof.* In a failure-free setting, whenever a node  $u$  forms a new connection, it increases its *degree* variable by one, and whenever it deactivates a connection, it decreases it by one. In case of a fault, all the adjacent nodes are notified, as their *fault flag* becomes one. Thus, they decrease their *degree* by one. In case of a fault with no adjacent nodes, a random node is notified, and its *fault flag* becomes two. In that case, it leaves the value of *degree* the same.  $\square$

**Theorem 11.** *For any N-NET protocol  $\Pi$ , it holds that  $(B, \Pi)$  is a fault-tolerant version of  $\Pi$ .*

*Proof.* Consider the case where a node  $u$  (either leader or follower) needs to be restarted. It enters to the restarting phase in order to deactivate all of its enabled connections, and it will start executing  $\Pi$  only after its degree becomes zero (by Lemma 12 this will happen correctly), thus,  $\Pi$  always run in nodes with no spurious edges (edges that are the result of previous executions). Whenever two connected nodes  $u \in R$  and  $v \notin R$  interact with each other, they both decrease their *degree* variable by one, and they delete the edge joining them. Obviously, this fact interferes with the execution of  $\Pi$  in node  $v$  (which is not in the restarting phase), but  $v$  is surely in a previous phase than  $u$  and will eventually also enter in  $R$ . This follows from the fact that a node in some phase  $i$  can never start forming new edges before it has successfully deleted all of its edges before. New edges are only formed with nodes in the same phase  $i$ .

The new *Restarting Phase* does not interfere with the states of the *PP Restarting Protocol*, thus the correctness of  $B$  follows by Lemma 9 and Lemma 10.  $\square$

**Lemma 13.** *The required memory in each agent for executing protocol  $B$  is  $O(\log n)$  bits.*

*Proof.* The maximum value that the variable *degree* can reach is the *maximum reachable degree* ( $d$ ) of protocol  $\Pi$ . Thus, by Lemma 11, the states that each node is required to have is  $O(dkn)$ . Both  $d$  and  $k$  are less than  $n - 1$ , thus,  $O(n^3)$  states =  $O(\log n)$  bits.  $\square$

## 5 Conclusions and Open Problems

We show that the class of properties for which fault-tolerant protocols exist in the SNET model is relatively small. Thus, it is imperative that we enhance this model so as to widen the class of such properties. In Sections 4.1 we give protocols that tolerate any number of faults, by introducing fault notifications. In Section 4.2, we present a fault-tolerant protocol which constructs a spanning line, and we show that it is able to simulate a given TM  $M$ . We then build upon that, and we present a generic fault-tolerant constructor capable of constructing any graph language that can be decided by a linear space TM. It operates by repeatedly constructing random graphs on the half of the population, while the other half executes the TM. We then show that if the constructed network is required to occupy  $1/3$  instead of half of the population, the available space of the TM  $M$  dramatically increases to  $O(n^2)$  from  $O(n)$ . We provide a fault-tolerant protocol which constructs any graph language  $L$  that can be decided by  $M$ . In both generic constructors, we assume that the nodes are capable of tossing an unbiased coin during an interaction. Can we drop this assumption? For example, this can be done by enumerating all possible graphs that can be constructed in  $D$ . If the TM has enough memory to do so, then this assumption can be dropped. Note that our results on fault-tolerant universal construction introduced some waste in the population. An interesting open problem is whether we can achieve a better trade-off between waste and constructive power of the TM.

If we do not allow waste, for protocols with *unbounded reachable degree* and nodes with constant number of states, we show impossibility of tolerating a single crash fault, even when the whole population is notified about that fault. Thus, we need additional assumptions, such as logarithmic memory ( $\log n$  bits). Under this assumption, we give a fault-tolerant protocol which correctly restarts any Population Protocol (with no connections between the nodes), and then we extend this in order to handle any N-NET protocol.

We have partially characterized the class of properties that fault-tolerant SNET protocols exist. We show that for any property in class  $PG \subset \text{Hereditary}$ , such protocols exist, and that for any property in  $\overline{\text{Hereditary}}$ , there are no fault-tolerant protocols that tolerate even a single crash failure. Even though we show a fault-tolerant protocol that constructs a graph property  $P \in \text{Hereditary} \setminus PG$ , the exact characterization remains an open problem. In addition, an interesting open problem is to study the stabilization time of fault-tolerant protocols (in both SNET and N-NET models). To this end, a reasonable measure of time needs to be formally defined. For instance, one could count the time to stabilization after the final fault. This can be used to measure the efficiency of fault-tolerant protocols. In this work, we considered only crash failures. Other immediate open questions are cases of random and Byzantine faults. Finally, a major open front is the examination of fault-tolerant protocols for self constructing *dynamic networks* in models stronger than the Network Constructors.

## References

- [1] Othon Michail and Paul G. Spirakis. Simple and efficient local codes for distributed stable network construction. *Distributed Computing*, 29(3):207–237, 2016.
- [2] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, March 2006.
- [3] Othon Michail, Ioannis Chatzigiannakis, and Paul G. Spirakis. Mediated population protocols. *Theoretical Computer Science*, 412(22):2434–2450, May 2011.
- [4] Ryu Mizoguchi, Hirotaka Ono, Shuji Kijima, and Masafumi Yamashita. On space complexity of self-stabilizing leader election in mediated population protocol. *Distributed Computing*, 25(6):451–460, 2012.
- [5] Giuseppe Antonio Di Luna, Paola Flocchini, Taisuke Izumi, Tomoko Izumi, Nicola Santoro, and Giovanni Viglietta. Population protocols with faulty interactions: the impact of a leader. In *International Conference on Algorithms and Complexity (CIAC)*, pages 454–466. Springer, 2017.
- [6] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1):3–16, Nov 1993.
- [7] Shlomi Dolev. *Self-stabilization*. MIT Press, Cambridge, MA, USA, 2000.
- [8] Nabil Guellati and Hamamache Kheddouci. A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs. *Journal of Parallel and Distributed Computing*, 70(4):406 – 415, 2010.
- [9] Bertrand Ducourthial and Sébastien Tixeuil. Self-stabilization with r-operators. *Distributed Computing*, 14(3):147–162, Jul 2001.
- [10] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Eric Ruppert. When birds die: Making population protocols fault-tolerant. In *IEEE 2nd Intl Conference on Distributed Computing in Sensor Systems (DCOSS)*, volume 4026 of *Lecture Notes in Computer Science*, pages 51–66. Springer-Verlag, June 2006.
- [11] Rachid Guerraoui and Eric Ruppert. Names trump malice: Tiny mobile agents can tolerate byzantine failures. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 484–495. Springer, 2009.
- [12] Dana Angluin, James Aspnes, Michael J. Fischer, and Hong Jiang. Self-stabilizing population protocols. *ACM Trans. Auton. Adapt. Syst.*, 3(4):1–28, 2008.
- [13] David Peleg. As good as it gets: Competitive fault tolerance in network structures. In Rachid Guerraoui and Franck Petit, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 35–46, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.